Gordon Bell Prize Lectures

J. J. Dongarra[1], A. Karp[2], K. Miura[3], and H. D. Simon[4]

Report RNR-91-023, August 1991

NAS Systems Division
Applied Research Branch
NASA Ames Research Center, Mail Stop T045-1
Moffett Field, CA 94035

August 12, 1991

Abstract. The Gordon Bell Prize recognizes significant achievements in the application of supercomputers to scientific and engineering problems. This is a summary of the entries in the 1990 competition, and a detailed discussion of the accomplishements of the winning entries.

(to appear in the Proceedings of Supercomputing '91, Albuquerque, November 18 - 22, 1991)

[1]Department of Computer Science, University of Tennessee, Knoxville, TN 37996.

[2]IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304.

[3]Fujitsu America Inc., 3055 Orchard Dr., San Jose, CA 95134.

[4]The author is an employee of Computer Sciences Corporation. This work is supported through NASA Contract NAS 2-12961.

Gordon Bell Prize Lectures

J. J. Dongarra
Oak Ridge Natl. Lab
Dept. of Comp. Sci.
Univ. of Tennessee
Knoxville, TN 37996

A. Karp
IBM Scientific Center
1530 Page Mill Road
Palo Alto, CA 94304

K. Miura
Fujitsu America
3055 Orchard Dr.
San Jose, CA 95134

H. D. Simon
Mail Stop T045-1
Comp. Sciences Corp.
NASA Ames Res. Center
Moffett Field, CA 94035

Abstract

The Gordon Bell Prize recognizes significant achievements in the application of supercomputers to scientific and engineering problems. In this special session the winners of the 1990 prize will give presentations about their winning entries in the competition.

1 Introduction

The Gordon Bell Prize recognizes significant achievements in the application of supercomputers to scientific and engineering problems. In 1990, two prizes were offered in three categories: performance, price/performance, and compiler parallelization. The performance prize recognizes those who solved a real problem in less elapsed time than anyone else. The price/performance prize encourages the development of cost-effective supercomputing. The compiler prize encourages the development of smart, parallelizing compilers.

In the past years the award ceremony for the Gordon Bell Prize took place at the Compcon conference, which is usually held at the end of February in San Francisco. This is a well attended general computing conference, but it is not in particular focused on high performance computing. Since the Gordon Bell Prize has received increasing public attention, a special session has been organized at Supercomputing '91, in order to give the winners of the prize a better forum for presenting their work. It is intended to continue this tradition and organize similar sessions at future Supercomputing X conferences.

2 The 1990 Competition

We received twelve entries – seven were considered for the performance prize, two for price/performance, and two for compiler generated speed-up. One enterprising entrant, knowing how flexible the rules are, entered in a non-existent category, speed-up. Prizes were awarded in the price/performance and compiler generated speed-up categories. Honorable mentions were named in the performance and compiler generated speed-up categories. Gordon Bell, vice president of engineering at Ardent Computer in Sunnyvale, Calif., is sponsoring two $1,000 prizes each year for 10 years to promote practical parallel-processing research. This is the fourth year of the prize, which IEEE Software administers. The winners were announced February 27, 1991 at the Computer Society's Compcon conference in San Francisco.

A check for $1,000 went to G. A. Geist and G. M. Stocks of Oak Ridge National Laboratory, B. Ginatempo of the Universitá of Messina, Italy, and W. A. Shelton of the Naval Research Laboratory for winning the price/performance award. They computed the electronic structure of a high temperature superconductor on a 128-node Intel iPSC/860 at a price/performance of over 0.8 Gflops/$1 million. In addition, they solved the same problem on a network of 11 IBM RS/6000 workstations at 0.7 Gflops/$1 million. (Four of the least expensive of these machines performed at 1.9 Gflops/$1 million.) The 2.5 Gflop rate for the run on the Intel machine is the first report of the Gflop barrier being broken by a multicomputer running a real application.

Gary Sabot, Lisa Tennies, Alex Vasilevsky of Thinking Machines Corporation and Richard Shapiro of the United Technologies Corporation received $500 for their work in compiler generated speed-up. They used a Fortran 77 to Fortran 90 conversion package to parallelize a grid generation program used to solve partial differential equations. They achieved a speed-

up of 1,900 and ran at over 1.5 Gflops on a Connection Machine with 2,048 floating point processors.

Two honorable mention prizes of $250 each also were awarded. The winners of last year's performance award, Mark Bromley, Steven Heller, Cliff Lasser, Bob Lordi, Tim McNerney, Jacek Myczkowski, Irshad Mufti, Guy L. Steele, Jr., and Alex Vasilevsky of Thinking Machines Corporation submitted the same application code that won last year. They increased the size of the problem and used an improved compiler and library. Just these changes enabled them to improve their job's performance by more than 2.5 times to 14 Gflops on a 64K processor CM-2G.

The other honorable mention goes to Eran Gabber, Amir Averbuch, and Amiram Yehudai of Tel-Aviv University for a parallelizing Pascal compiler. They successfully parallelized 14 programs, including one of the programs that won the first Gordon Bell Prize, achieving speed-ups of up to 25 on 25 processors of a Sequent Symmetry.

The judges felt that, although they did not win prizes, two entries were worthy of note. A seismic migration problem submitted by a group from the Colorado School of Mines headed by G. Almasi ran at over 1.5 Gflops/$1 million on a network of 4 IBM RS/6000s. David Strip of Sandia National Laboratories and Michael S. Karasick of IBM Research figured out how to do solid modeling on a Connection Machine over 35 times faster than they could on an IBM 3090 vector processor.

The price/performance category presented a new problem this year. In the past, all entries in this category ran on machines costing millions of dollars. Adding the cost of software, a display, and keyboard to that of one of these machines will not have a major impact on the price. This year three entrants used collections of workstations, and each made different assumptions about the equipment needed to run the jobs. For example, the Colorado School of Mines group used a price for the RS/6000 model 320 nearly twice that used by the Oak Ridge group. Both price/performance figures were adjusted, Colorado up substantially and Oak Ridge down slightly, to reflect the currently published prices.

3 The Price/Performance Winner

Theoretical materials scientists have traditionally studied either highly ordered materials such as crystals or highly disordered materials such as glasses. Recently, however, there has been a great deal of experimental work done on what are called "substitutionally disordered materials." The one that has received the most press attention is the family of high temperature superconductors, but metallic alloys and materials with metal-insulator or magnetic phase transitions also fall into this category.

These materials are messy, and interpreting the experimental data is difficult. Some five years after the discovery of high temperature superconductors we still do not have an understanding of why they lose electrical resistance. Thus, theoreticians have been called on to help. While semi-empirical studies can be useful, it is widely felt that we will only understand these materials if we can compute their observed properties from first principles. In this case, first principles means relativistic quantum electrodynamics.

It is not yet possible to solve the Schroedinger equation for a solid; there are just too many degrees of freedom. Instead, we use methods that enable us to average over the bulk properties, leaving a manageable number of states.

One approach is based on an observation arising from density functional theory. It states that one of the fundamental properties of a solid, its ground state energy, depends uniquely on the electron density. Furthermore, the electron density can be computed by solving for the motion of a single electron moving through an electric field that is an average of that of all the nuclei and other electrons. This solution must include the effect that the moving electron has on all the other electrons. We say that the solution must be self-consistent. The complex, many-electron effects are approximated by treating the background electrons as a continuous gas and using the density of that gas at any point as an approximation to the true electron density, i.e., by making the "local density approximation."

The local density approximation does not work for substitutionally disordered materials. Instead, it is necessary to use a method such as the coherent potential approximation. The effects the disordered crystal has on the electronic structure are approximated by those of some "effective" scatterer. An ordered, periodic array of effective scatterers gives, in some sense, the best approximation to the disordered structure of the material. The coherent potential approximation computes the best effective scatterer that can be obtained using only average properties of the nuclei.

Finding the effective scatterer involves the self-consistent solution of a set of integral equations. The procedure starts with a guess to the electron distribution. Next, a large number of numerical quadratures are done. Any deviations from self-consistency are

used to update the electron distribution. The procedure is continued until it converges.

The integrals representing the solution are done over the fundamental modes of the crystal which depend on the locations of all the nuclei. If the algorithm were parallelized at this level, each processor would be responsible for a subset of the crystal. Since, these integrals involve data from the entire crystal, this scheme would require a lot of communication among processors. Fortunately, the function being integrated over the fundamental modes involves an integral over the energy which can be done separately at each atomic site.

The integration over energy requires between 200 and 1,000 energy evaluations at each site to determine the charge distribution for the next iteration. Each energy evaluation involves the iterative solution of the coherent potential approximation equations. Since there is a great deal of computation and little communication, this approach is well-suited for parallel processing, even on a loosely-connected set of independent processors.

The parallelization was implemented using a master/slave approach. One processor is responsible for reading the problem description, the location of the input files, and managing the overall iteration. Load balancing is achieved by assigning tasks to processors in order of decreasing difficulty.

An interesting feature of the code is its portability; it has versions for serial, shared memory, and distributed memory computers. The multicomputers can be moderately closely coupled systems like hypercubes, or a loosely connected collection of workstations. The input file determines whether multitasking or message passing will be used. The total program is 16,000 lines of Fortran and contains 127 subroutines. Of these, only about 20 are explicitly involved in the parallelism.

Results were presented for the simplest of the Perovskite superconductors, a family that includes several of the other high temperature superconductors. This Barium-Bismuth system is of interest because it has only five simple cubic sublattices, it has a cubic symmetry that reduces the amount of computation, and it is closely related to the more complicated systems involving lanthanum and yttrium that have received a lot of attention.

Most of the time in this calculation goes into constructing a 160 x 160, dense, complex matrix (12%) and then inverting it (78%). These matrices are constructed and inverted independently on each node hundreds of times per iteration. The matrix construction runs at over 24 Mflops per i860 node, 3 Gflops aggregate, while the inversion is done at 21 Mflops, 2.6 Gflops aggregate. Each processor of a Cray Y-MP does these tasks at 300 Mflops which translates into a 2.4 Gflop aggregate rate.

A complete calculation involving almost 4×10^{13} floating point operations took 4.5 hours on a 128 node Intel iPSC/860, a computational rate of 2.5 Gflops. Since the machine being used has a list price of $3 million, this performance translates into a price/performance of over 0.8 Gflop/$1 million. This rate includes all the start-up time and load imbalance.

This same program was ported to a network of IBM RS/6000s. A single RS/6000 model 320 having a list price of $8,500 ran at 16.7 Mflops, which corresponds to 2.1 Gflop/$1 million. Although this price performance is impressive, it would have taken about a month to complete one model. However, four of these bottom of the line machines could finish the job in about a week at a price performance of over 1.9 Gflop/$1 million. Since neither of these configurations could complete the full problem in a reasonable amount of time, a run was made on all the RS/6000s available at Oak Ridge. This set of 7 model 530s and 4 model 320s ran at a sustained rate of 226 Mflops and a price/performance of over 0.7 Gflop/$1 million. Note that the speed-up on this application is nearly linear in the number of processors. Therefore, if there had been 11 model 320s at Oak Ridge these figures would have been 170 Mflops and a price/performance of about 1.8 Gflop/$1 million.

4 The Compiler Speed-up Winner

As engineers have improved their models, they have found it increasingly necessary to work with complicated shapes. Those who wish to use finite difference or finite element methods must construct grids that match the complicated boundaries associated with realistic models. Unfortunately, carelessly generated grids result in solutions with lots of error. A great deal of effort has gone into generating grids that do a good job of following the objects being modeled and having desirable numerical properties.

Generating a "good" grid is nontrivial. Consider the simple problem of modeling the flow over a ramp. A grid consisting of uniformly spaced lines in the vertical and horizontal directions is unlikely to be satisfactory. The grid intersections will not necessarily fall on the ramp which will make it difficult to handle the boundary conditions. Furthermore, the ends of the

ramp are likely to be places needing higher resolution than the regions on either side of the ramp.

A better approach might be to produce a grid that is locally perpendicular to the surface. While the boundary can be represented accurately this way, the grid can become highly non-uniform. The sudden changes in grid spacing and the odd shapes of the grid cells can seriously degrade the numerical accuracy of the partial differential equation solver.

Numerical methods are often rated on how quickly the error in the solution decreases as the grid is refined. For example, in two dimensions a good method with a good grid will reduce the error by a factor of 4 when the spacing is halved. However, even a good method will reduce the error only linearly as the spacing is refined if the grid is poor. If the grid is very bad, the convergence can be sublinear.

While it is difficult for a person to design a good grid even for something as simple as a ramp, it is virtually impossible to build a good grid for the kinds of surfaces engineers are interested in, things like the space shuttle or the Rocky Mountains. Thus, some sort of automatic tool is needed.

A grid generation program is such a tool. It produces a grid that conforms to the shape being modeled and retains the good numerical properties of the discretization. Such a grid has a spacing that changes smoothly and has grid lines that cross at nearly right angles. The result is a grid that conforms to the shape being studied and with grid points lying on smooth curves.

Conceptually, the grid generation algorithm warps the surface into the unit cube and places a uniform grid either around this cube if we are modeling the exterior or inside it if we are interested in the interior. The differential equation solver is then applied to this simple domain. To visualize the results we then transform back into the original coordinates.

This warping is a coordinate transformation determined by solving a system of partial differential equations. The solution of these equations gives us the coordinates of the intersections of the new grid lines. There are many ways of affecting the transformation because the properties of a "good" grid, such as skewness and relative size of neighboring grid cells, are specified so inexactly.

The various grid generation approaches differ primarily in their choice of the equations used to generate the grid. The method used in the winning entry starts with an arbitrary grid that follows the boundary. A system of second order, non-linear differential equations that represents the coordinates of the new grid as functions of the coordinates of the old grid is used to compute the transformation. There is one differential equation for each coordinate direction (two for the 2D problem submitted), each involving all combinations of second derivatives. The coefficients of the second derivatives involve only first derivatives of the new coordinates with respect to the original ones. It is these coefficients that make the problem nonlinear.

The solution of these equations is necessarily an iterative process. First, the differential equations are discretized using standard central difference approximations for the first and second derivatives. The resulting system of nonlinear, algebraic equations can be solved using a variety of methods, many of which do not parallelize well.

The model submitted was run on a data parallel Connection Machine, a CM-2G. This Single Instruction Multiple Data (SIMD) computer works well on algorithms that involve lots of independent data elements. One such algorithm, a relaxation method using a Jacobi-like iteration, was selected for the computations. This method works by guessing a solution, evaluating the discretized differential equation to compute the residual, and setting the new value of each grid coordinate to the old value plus the computed residual times a relaxation parameter. The relaxation parameter was used because the approximations oscillate around the solution as the iteration proceeds. While there are methods that converge in fewer iterations, they are more complicated, less parallel, or involve more floating point operations per iteration. Key to the judges accepting this entry is the fact that Jacobi-like iterations are commonly used, even on sequential, scalar computers.

The entrants took a grid generation program written in Fortran 77 containing no compiler directives. This program was passed through the KAP/F77-F90 preprocessor from Kuck and Associates. The preprocessor converted many of the nested loops in the original program into Fortran 90 array constructs. For this particular program, the sophisticated dependence analysis done by the KAP preprocessor was not needed; the important loops were very simple. The resulting Fortran 90 program was compiled with the CM Fortran compiler which applies conventional vectorization optimizations to parallelize the code. The two stage approach avoided the difficult question of whether the user or the compiler has determined the parallelism of a program written using Fortran 90 array constructs.

One of the difficulties the judges had with this entry was determining the meaning of speed-up on a

SIMD machine. In particular, should we count each of the 65,536 one-bit processors as a separate machine or only the 2,048 floating point chips? How can you make a run on one processor of a SIMD machine? Do we allow the problem to increase in size as the number of processors increases or will we consider only fixed-size speed-up?

The original submission based its speed-up measurement on the time of the original Fortran 77 program running on the CM-2 front-end machine, a Sun 4 with a Weitek floating point chip. Although this floating point chip is similar to the one used on the CM, the judges questioned the validity of the reported speed-up of 4,900, especially since the version of the CM Fortran compiler used treated the machine as 2,048 nodes and not 65,536 processors. Clearly, the floating point performance of the Weitek chip on the Sun is not simply related to that on the CM-2.

The judges asked the entrants if they could run the job on one of the nodes. They did this by changing one environment variable to fool the system into thinking that the machine consisted of just one node containing a single, 64-bit FPU and 4 megabytes of memory. (They were quite surprised when this approach worked since the compiler testing strategy had not covered this serial configuration of the CM). Then they ran the largest problem that could fit in the memory connected to this node.

The fundamental unit of computation was the number of grid points processed per second. Since this quantity is relatively insensitive to the size of the problem, we felt it was reasonable to compare the 128×128 grid on one floating point unit to the 8192×4096 grid computed on the full machine. This method resulted in a speed-up of 1,900. Equally impressive was the 2.3 Gflops computation rate.

5 Honorable Mentions

The first of the honorable mentions goes to the group from Thinking Machines Corporation that won the performance prize for 1989. They improved the performance of the application they submitted last year by a factor of 2.5 by throwing away last year's low level hand code and applying improved compiler technology to the original Fortran 90 source code. Many of these compiler improvements were produced by the same Thinking Machines compiler group that won this year's prize for compiler generated speed-up!

The 14 Gflops they report is truly remarkable, as is their price/performance of approximately 1.4 Gflop/$1 million. The CM-2G is the most expensive machine to achieve over 1 Gflop/$1 million on a real problem. Because the problem submitted is identical to the one that won last year's performance prize, the description that follows concentrates on the changes made to the software.

The CM-2G is a data parallel, Single Instruction Multiple Data (SIMD) computer containing 65,536 one-bit processors, 8 Gbytes of memory, and, optionally, 2,048 64-bit floating point processors. Data parallelism is used to program the CM-2. Basically, the programmer acts as if each data element, say a grid point in a finite difference solution, is processed by a separate processor having its own private memory.

One hardware change is important, the increased memory. In general, the larger the problem, the higher the performance. The main reason for this behavior is that communication between points assigned to a single floating point unit is faster than between those assigned to different ones. The four-fold increase in the memory between the CM-2 used last year and the CM-2G used this year allowed the entrants to run a larger problem.

The primary change made in the software is the compiler's view of the target machine. Each 64-bit floating point unit and its associated 4 Mbyte of memory is called a "node" and acts much like a vector processor. The CM is viewed as consisting of 2,048 64-bit vector processor nodes instead of a collection of 65,536 one-bit processors. Loop spreading and strip mining are used to map problems that have more points than 2048 times the vector length (currently 4) onto the machine.

This change is reflected in the way CM Fortran stores floating point numbers. The previous release of the compiler stored all 32 bits of each floating point number on a single, bit-serial processor. Unfortunately, this approach does not mesh well with the Weitek floating point chips.

On a given machine cycle each CM processor can access one-bit from its local memory. It takes, therefore, 32 memory cycles to access a floating point number. The floating point unit, on the other hand, expects to get a 32 bit number on every cycle. This problem was handled in hardware by inserting a transposer between the CM processors and the floating point unit. Over a period of 32 cycles, the transposer takes in 32 floating point numbers, one from each of the 32 one-bit processors. It then feeds these numbers, one per cycle, into the floating point unit. The process is reversed when the output is to be stored. While this approach does allow the pipelining of the Weitek chip

to be used, it impedes the overall performance of the machine.

The newest release of the CM Fortran compiler stores the data "slicewise". In this storage scheme, each bit of a floating point number is stored on a different one-bit processor. Now, on one cycle 32 bits of the number can be sent directly to the floating point unit, bypassing the transposer array. (Two cycles are used for 64-bit numbers.) Slicewise data storage also changes the programmer's view because a problem with as few as 2048 grid points will use all 2048 64-bit processors, replacing the view where 65,536 points were needed to use the whole machine.

This new view of the hardware was extended to the communications. The older library was set up so that each of the one-bit processors would pass a single piece of data to a specific neighbor. Since all the data moved in the same direction on each step, it would take 4 communications steps to distribute data in a two dimensional grid. The newly microcoded communications primitives allow nodes, as opposed to processors, to communicate on the two dimensional grid. There are enough wires from each node to allow them to pass data in all four directions in one communications cycle.

Another improvement to the compiler is the way it handles fundamental grid operations. The finite difference discretization of a partial differential equation involves several grid points. The particular choice of grid points for a given discretization is called a "stencil." In two dimensions the discretization which combines each point with its North, South, East, and West neighbors is called the 5-point stencil. Other, more complicated stencils, are also used.

Last year Thinking Machines introduced a stencil library that used highly tuned microcode to improve the performance of the communications of some commonly used stencil calculations. This year, a compiler was written which automatically recognizes any kind of stencil, even an irregular one. Basically, any CM Fortran statement of the form

$$\mathtt{A} = \mathtt{C1} * \mathtt{S(X)} + \mathtt{C2} * \mathtt{S(X)} + ..$$

can be compiled as a stencil, as long as `A` and `X` are arrays and `S` is any of the functions that shift data in the grid. The stencil compiler uses a number of special coding tricks to improve performance. For example, at the start of the computation all the data needed by the node will be copied into the node's local memory. Furthermore, optimizations can be used to make better use of the floating point chip's registers and pipelines. Every effort has been made to minimize the movement of data between the floating point unit and the node's memory.

The other honorable mention is the first cash award made for work done in Pascal. The compiler this group from Tel-Aviv University wrote is remarkable in several respects. The same compiler can be used for both shared memory and distributed memory machines. Furthermore, no directives are needed to help the compiler distribute the data among processors.

The input language, Tel-Aviv University (TAU)-Pascal, includes only a few, minor extensions to Pascal. The Portable Parallelizing Pascal Compiler (P^3C) consists of a front-end which takes a sequential, TAU-Pascal program and produces an explicitly parallel C code for Virtual Machine for MultiProcessors (VMMP). VMMP is a software package that implements a set of parallel processing functions for a diverse set of machines. P^3C will produce good parallel code for any machine that can run VMMP. The data partitioning and code generation are optimized for the target machine by consulting a table of parameters that specifies the relative cost of certain operations, such as floating point arithmetic or communication.

The P^3C compiler produces a parallel program that runs in Single Program Multiple Data (SPMD) mode. This means that each of the parallel processes executes the same lines of code on different parts of the data. Some parts of the program are executed by a subset of the processors by using the task id to control statement execution. For example, program initialization and termination are handled by a single processor. The task id is also used to split up work on an array among the processors.

P^3C does not use the most sophisticated dependence analysis to determine whether a loop can be parallelized safely. A loop will be parallelized only if the loop is such that each iteration modifies a different array element or if the loop contains certain simple grid operations. The loop may also be parallelized if it contains certain reduction operations, such as inner product, or specific access patterns on columns of matrices. The current version will not parallelize a loop nested within a previously parallelized loop.

P^3C supports only a limited set of data partitionings. These are replication (each process gets a complete, read-only copy), replication for data reduction (certain associative and commutative operations are allowed on the copies), row distribution (each process gets a contiguous block of rows), overlapped row distribution (each process gets a contiguous block of rows with some duplication between the processes owning neighboring blocks), interleaved row distribution (the

rows are dealt out as in a card game), and interleaved column distribution. As limited as these patterns are, they are sufficient to parallelize a wide variety of programs.

The Tel-Aviv group submitted 14 different programs, each run on three very different parallel systems - a Sequent Symmetry shared memory machine with 26 processors, MOS, an experimental shared memory system with 8 NS32532 processors, and a network of 8 T800 Tranputers.

The application set is quite diverse. It includes computational kernels, like the solution of dense systems of linear equations by conjugate gradients and matrix multiplication, to such applications as a particle in cell simulation of electron beams and the simulation of 150 bodies attracted by gravitational forces. The judges worried about how realistic these codes were, but we were convinced when P^3C was used to parallelize the WAVE program that won the first year's Bell Prize. In all cases, P^3C generated code ran with between 60% and 99.8% efficiency on all three parallel systems.

The judges ran into a unique problem with this entry. As typically happens, we needed some questions answered before we felt comfortable awarding a prize. Unfortunately, just at the time we needed a quick response to meet our deadline, Tel-Aviv was under attack from Iraqi missiles! Fortunately, someone from the Tel-Aviv group managed to get out long enough to answer our e-mail questions.

6 Other Entries

The Gordon Bell Awards are attracting entries in many areas, many of which we did not anticipate. An entry from I. Beichel and F. Sullivan of the National Institute of Standards and Technology demonstrated the first robust and efficient code for 3 dimensional triangulation, an important part of some molecular dynamics simulations. A. K. Lenstra of Bellcore, H. W. Lenstra of UC Berkeley, M. S. Manasse of DEC in Palo Alto, CA, and J. M. Pollard of Reading, England submitted their widely publicized factorization of the ninth Fermat number, an integer with 155 digits.

The remaining entries were more along the lines we were expecting. L. A. Feldman of the US Air Force and B. M. Dodd of Cray Research submitted a calculation of the incompressible flow around an F-16 aircraft flying at Mach 1.8 that only took 30 seconds to complete. P. Highnam, A. Pierprzak, and I. Chakravarty of Schlumberger and B. Moorhead, C. Liu, B. Biondi, and D. Race of Thinking Machines Corporation did a seismic migration problem on a CM-2. W. C. Liu, P. Rossi, and M. Bromley of Thinking Machines Corporation and A. D. Kennedy, R. Edwards, and D. Sandee of Florida State University used a quantum chromodynamics program running on a CM-2 to attack one of the "grand challenge" problems of computational science. P. Emeagwali, one of last year's winners, submitted his version of the NCAR shallow water model running on a CM-2.

A new rule is being added for the 1991 Gordon Bell Prize. In addition to prizes for performance, price/performance, and compiler generated speed-up, a category is being added to recognize those who use parallelism in a significant way to make the largest improvement in solving important problems. Details will be published at a later date.

7 Program for the Special Session at Supercomputing 1991

Session Chair: Horst Simon, Computer Sciences Corporation at NASA Ames Research Center.
Presentations:

1. *A History of the Gordon Bell Prize*, by Alan Karp, IBM Scientific Center, Palo Alto, California. (10 min) (A summary of the history of the award, entries for the 1990 award, rules and plans for the 1991 awards.)

2. *A Portable Parallelizing Pascal Compiler for Shared and Distributed Memory Multiprocessors*, by Eran Gabber, Amir Averbuch, and Amiram Yehudai, Tel-Aviv University, Israel; honorable mention; 12 min + 3 min for questions; presented by Eran Gabber.

3. *Seismic Modeling at 14 Gigaflops on the Connection Machine* by Mark Bromley, Steven Heller, Cliff Lasser, Bob Lordi, Tim McNerney, Jacek Myczkowski, Guy L. Steele Jr., and Alex Vasilevsky, Thinking Machines Coproration; honorable mention; 12 min + 3 min for questions; presented by Guy Steele.

4. *Compiler Parallelization of an Elliptic Grid Generator*, by Gary Sabot, Lisa Tennies, and Alex Vasilevsky of Thinking Machines and Richard Shapiro of United Technologies; compiler speed-up prize $500; 20 min + 5 min for questions; presented by Gary Sabot.

5. *Parallel Superconductor Code Exceeds 2.5 Gflops on iPSC/860*, by A. Geist and G. Stocks of Oak Ridge National Laboratory, B. Ginatempo of the University of Messina, Italy, and W. Shelton of Naval Research Laboratory, price-performance prize $1000; 20 min + 5 min for questions; presented by Al Geist

8 Abstracts of the Talks

8.1 History of the Gordon Bell Prize

Alan Karp
IBM Scientific Center
1530 Page Mill Road
Palo Alto, CA 94304

Abstract. At the 1985 SIAM conference on parallel processing there was a lot of talk about building 1,000 processor, 10,000 processor, and even 1,000,000 processor systems. However, no one had shown that reasonable speed-ups could be obtained even on much smaller systems. Tired of hearing lots of talk and seeing no demonstrations, in 1986 Alan Karp offered $100 for the first person to demonstrate a speed-up of at least 200 on a real problem running on a general purpose parallel processor. The offer was to last for 10 years.

Gordon Bell thought the challenge was a good idea, but he did not think anyone would win it. To keep things interesting for the duration of the challenge, he offered his own prize - $1,000 for the best speed-up of a real application running on a real machine. An additional category was added for special purpose machines.

Bell had three goals in mind for his prize.

- Reward practical use of parallel processors.
- Encourage improvements in hardware and software.
- Demonstrate the usefulness of parallel processors for real problems.

These goals remain the same 5 years later.

As is now well known, the winners of the first Bell Prize took home all the marbles, Bell's money as well as Karp's. The group from Sandia National Laboratory demonstrated a speed-up of over 400 on three large applications running on a message passing hypercube with 1,024 processors. In addition, they pointed out that real users run larger problems on faster machines. They calculated speed-ups of close to 1,000 if the problem size scaled with the number of processors.

The rules were modified following the first year and have remained the same since then. Each year the judges have $2,000 to divide among two winners and any honorable mentions. The recipients are selected from entries in three categories – performance, price/performance, and compiler generated speed-up.

Performance is normally measured in millions of floating point operations per second (Mflops). Entrants must convince the judges that they have run their application faster than anyone else has done. Although the judges end up comparing the raw performance of a wide variety of applications, each year a clear winner has emerged.

The price/performance prize encourages the development of cost effective supercomputing. The rules are set up to prevent unrealistic machines from winning the prize. For example, a parallel job running at 1 Kflop per second on two used Z-80 processors with a price of $1 is not eligible. (Interestingly enough, such a machine would not have won a prize this year.)

The compiler generated speed-up prize is intended to spur the development of compilers that can automatically parallelize sequential programs. This problem is a difficult one as evidenced by the fact that this category received its first prize only this year.

No toy problems or cooked up examples are allowed; we are looking for real problems run on real machines. Although the rules are quite flexible, it is up to the entrants to convince the judges of the quality of the work submitted.

Four sets of prizes have been given. The original rules rewarded large speed-up. Subsequently, the rules were modified to emphasize performance and price/performance. The progress in those 4 years has been remarkable. Examination of the Table shows that, while speed-up has stayed near 1,000, both performance and price/performance have improved dramatically.

Year	Performance Gflops	Price/perf. Gflops/$1 million	Speed-up
1987	0.45	0.03	600
1988	1.0	0.05	800
1989	6.0	0.5	1,100
1990	14.0	2.0	1,800

Manufacturers have announced some interesting machines, and software developers are promising easier to use tools and more sophisticated compilers. Let's hope that the rate of improvement continues.

8.2 A Portable Parallelizing Pascal Compiler for Shared and Distributed Memory Multiprocessors

Eran Gabber, Amir Averbuch, and Amiram Yehudai
Computer Science Dept.
School of Mathematical Sciences
Tel-Aviv University
Tel-Aviv 69978 ISRAEL
E-mail: eran@MATH.TAU.AC.IL

Abstract. The Portable Parallelizing Pascal Compiler (P^3C) is a research compiler, which translates serial programs written in a Pascal based language into a portable and efficient parallel code. P^3C generates code for both shared memory and distributed memory multiprocessors.

P^3C performs automatic data and process partitioning, which are guided by an accurate execution time estimation. The execution time estimation is performed by a static analysis of the program with reference to an external target parameters table. P^3C recognizes common data access patterns, such as for grids, arrays and data reduction, and generates efficient partitioning for them. P^3C is fully automatic and does not require any user directives or declarations to assist the partitioning.

P^3C consists of two parts: a target independent parallelizer and the virtual machine VMMP, which is a portable software environment running on diverse multiprocessors. P^3C can be ported easily to other multiprocessors by porting VMMP and generating the target parameters table.

P^3C has been implemented on two shared memory multiprocessors (a Sequent Symmetry and an experimental machine) and on a distributed memory multiprocessor (a network of Transputers). Porting P^3C to other multiprocessors should take 2-3 man months by our experience.

P^3C has been used to parallelize 14 application programs from diverse areas, including physical simulations, linear algebra kernels, cellular automata and PDE solution. One of these programs is the `Wave` program, which received the 1987 Gordon Bell Prize.

The parallel code produced by P^3C achieved speedup up to 24 on 25 processors of a Sequent Symmetry (shared memory), and up to almost 8 on a network of 8 Transputers (distributed memory). This speedup was achieved for cellular automata simulation and Monte-Carlo integration. The `Wave` program achieved a speedup of 22 and 7.8, respectively. Other programs with less than perfect parallelism achieved lower speedups. For example, matrix inversion achieved a speedup of 17.3 and 6.4, respectively.

8.3 Seismic Modeling at 14 Gigaflops on the Connection Machine

Mark Bromley, Steven Heller, Cliff Lasser, Bob Lordi, Tim McNerney, Jacek Myczkowski, Guy L. Steele Jr., and Alex Vasilevsky
Thinking Machines Coproration

Abstract. Seismic modeling represents a difficult numerical challenge and consumes a significant amount of CPU time on the largest available supercomputers. With the advent of massively-parallel supercomputers, there is a possibility of drastically reducing the execution time for some of these codes. Many of the algorithms used in seismic modeling use explicit numerical methods on regular structured grids. Because of the regularity of the interconnections and the locality of the calculations, those types of problems usually map well onto massively parallel computers. In this paper the acoustic wave equation with sponge boundary conditions will be used as an example to show how to map and optimize an explicit finite difference algorithm onto a massively parallel machine. This algorithm is part of a seismic modeling code developed jointly by Mobil Research and Thinking Machines to run on a CM-2 Connection Machine. This program achieved a sustained performance of 14.1 billion numerical operations per second (14.1 Gigaflops) including I/O on a 65536 processor CM-2 supercomputer.

8.4 Compiler Parallelization of an Elliptic Grid Generator

Gary Sabot, Lisa Tennies, and Alex Vasilevsky
Thinking Machines Corporation
and Richard Shapiro
United Technologies

Abstract. The paper presents a case study involving the application of an automatically parallelizing compiler to a numerically intensive Fortran program, and includes a detailed analysis of the result. The paper is based on work performed as part of an entry that won the 1990 Gordon Bell Prize for Compiler Parallelization.

The compiler involved is the CM Fortran 1.0 compiler, which targets the 2048 FPUs available in a full size CM-2 parallel computer. The application code implements grid generation for numerical simulation.

The paper presents a parallel speedup of 4770 over a serial run on a Sun workstation (which only has 1 FPU), and includes a cycle counting analysis of how such a speedup is possible with 2048 FPUs.

8.5 Parallel Superconductor Code Exceeds 2.5 Gflops on iPSC/860

G. A. Geist
Mathematical Sciences Section
Oak Ridge National Laboratory

B. Ginatempo
Istituto di Fisica Teorica
Unitá GNSM-CISM, Universitá of Messina, Italy

W. A. Shelton
Naval Research Laboratory
Washington, DC.

G. M. Stocks
Metals & Ceramics Division
Oak Ridge National Laboratory

Abstract. Researchers at Oak Ridge National Laboratory have developed a Fortran application code for calculating from first principles the electronic properties and energetics of disordered materials. The same source code has been compiled and run on workstations, Crays, and the Intel iPSC/860. This electronic structures code is capable of running over 2 Gflops on both an 8 processor Cray YMP and a 128 processor Intel iPSC/860.

Using this new KKR-CPA code, density-of-state computations of the high temperature perovskite superconductor $(Ba_{.6}K_{.4})BiO_3$ executed at a rate of 2527 Mflops on the Intel iPSC/860. This corresponds to a price/performance rate of 842 Mflops per $1 million based on the list price of this computer.

The KKR-CPA code was modified to use the software system PVM (Parallel Virtual Machine). PVM allows a network of heterogeneous computers to be utilized as a single computational resource. Many configurations of machines have been used in running the KKR-CPA code including a network of IBM RS/6000 workstations, a network of Cray supercomputers, and a Cray - Intel combination.

The highest price/performance ratio was 2 Gflops per $1 million and was achieved using a network of IBM RS/6000 model 320's. The highest raw performance observed so far is over 9 Gflops using a network of Crays.

This talk will describe the parallelization of the application code and the performance achieved on various machines.

Acknowledgements

The work by Eran Gabber, Amir Averbuch, and Amiram Yehudai was supported in part by the Basic Research Foundation administered by the Israel Academy of Sciences and in part by a research grant from National Semiconductor, Israel.

The research by G.A. Geist et al. was supported by the Applied Mathematical Sciences Research Program, Office of Energy Research, and the Division of Materials Sciences, U.S. Department of Energy, under contract DE-AC05-84OR21400 with Martin Marietta Energy Systems, Inc.

Horst D. Simon acknowledges support for his work through contract NAS 2-12961 with the National Aeronautics and Space Administration.

Portions of this report have been published previously in IEEE Software, May 1991, page 92.